A Simple Cutting Plane in WebGL

Rasmus Bons

Graphics Programming @ ITU, 2017

1 Introduction

In this project I have created an application where a defined plane, can be used to 'cut' slices off objects in a scene. The 'cut' reveals a clean surface rather than the backface, giving the illusion that the object is solid, without using voxels. The plane discards fragments in front of it, and uses the plane to calculate light. The plane also affect the shadow map generation, creating dynamic shadows for when the object changes. I have used WebGL2.0 to create this application.
In the next sections, I will further explain the theory of each element used in this project.

2 Theory

2.1 Plane Intersection

The core of this project is the plane intersection, which is two parts. First, the fragments on one side of the plane is discarded, and then the visible backfaces are evaluted in order to render correctly.

2.1.1 Discard fragments

In order to decide which fragments to discard, the cutting plane had to be defined in the shader. To optimize amount of data sent to the fragment shader, the plane is defined with a normal vector N and distance d taken from the scalar equation of the plane.

$$N_x \cdot x_p + N_y \cdot y_p + N_z \cdot z_p = d \tag{1}$$

Inside the fragement shader, every fragment is tested by taking the dot product of its worldspace coordinate and the planes normal vector. d is added to the dot product, and if the total sum is greater or equal zero, the fragment is on the normal direction side of the plane and is discarded. The exact same procedure is done in the depth fragment shader, which is used to create the shadowMap texture, in order to have consistency between the sliced models and their shadows.

2.1.2 Render backfaces

When the relevant fragments has been discarded, you would look onto backfaces of the model, which breaks the illusion of the object being whole. To uphold the illusion, all calculations for lights (modified phong model) and shadows of the backfaces were calculated based on the intersection point on the cutting plane, between the world position of the fragment and the world position of the camera.

The intersection point was calculated through the planes scalar equation and a parametric representation of a line for the two points

$$f(t) =< x_0, y_0, z_0 > +t < x - x_0, y - y_0, z - z_0 > \tag{2}$$

where $< x_0, y_0, z_0 >$is world position of the fragment, and $< x, y, z >$ is the camera position. From here, I insert the from the line representation into the planes equation so that,

$$\begin{aligned} x_p &= x_0 + (x - x_0)t \\ y_p &= y_0 + (y - y_0)t \\ z_p &= z_0 + (z - z_0)t \end{aligned}$$

Then, I solve for t:

$$t = \frac{d - (N_x \cdot x_0 + N_y \cdot y_0 + z_0)}{(N_x \cdot x + N_y \cdot y + N_z \cdot z_0)} \tag{3}$$

Finally, I can insert t into the parametric equation of the line, which gives the intersection point on the line, from which I use to e.g. calculate incident light(for diffuse) and halfway vector(for specular highlight).

2.2 Shadow Maps

For the shadows, I have used the technique 'Shadow mapping'. The main idea of the shadow map is to take a picture of the screen from the light sources point of view. To do that, you need create the a view and projection matrix based on the light source. I have created shadow map for both directional light and for point light(omni-directional light). In order to store the shadow map, I have used framebuffer objects. Frammebuffer objects are buffers that is created by the application, which is be render to rather than rendering to the screen. The render in this case is a texture. Based on each fragments corresponding texture coordinate(called shadow coordinates), the texture is used to look up depth. If the depth in the shadow map is lower than the fragments depth, then the fragment is in shadow.

2.2.1 Directional light

To simulate the directional light, I create a view matrix from a lookAt function, with eye at [0,0,0], lookAt target set to the direction of the light, and the up-vector of [1,0,0]. The projection matrix is orthogonal spanning from -7 to

7 from left to right and near to far, as directional light represent many light rays from the same direction. The size of the view frustum was defined through testing in order to get the best shadow map representation.
Since the depth in the shadow map is packed into values in the range of [0,1], the fragments position from the light is should also be in that range. To do so, I first convert the vertex position in the light projection from the range of normalized device coordinates(range[-1,1]), to [0,1]:

$$vPosFromLight.xyz = vPosFromLight.xyz \cdot 0.5 + (0.5, 0.5, 0.5) \quad (4)$$

then in the fragment shader, I do a perspective division in order to get the shadow coordinates

$$shadowCoords = vPosFromLight.xyz/vPosFromLight.w \quad (5)$$

The shadow coordinates z is checked against the unpacked depth of the shadow map texture. If the shadow coordinate is greater than the unpacked depth+bias, the fragment is in shadow and has its color reduced.

2.2.2 Omni-directional light

The basis of the shadow map is the same as for the directional light. However, rather than rendering to a single texture, the depth shader renders to a Cubemap. Each side of the cubemap has its own view matrix, representing each direction of the cubes faces. The projection matrix is identical for all sides, with 90 degrees Field-of-View, aspect ratio 1:1, near.plane 0 and far-plane 250 in order to collect everything around the light. Rather than having a single framebuffer and switching the images, I have chosen to have an array of six framebuffer objects that each have a side bound to it. This is done for optimization purposes.
For the omni-directional shadow map, the computation of shadowing fragments or not, differs from that of the directional light.

The vector from the light position to the fragment is converted from vector to depth value(function can be seen in VectorToDepthValue in SimpleCuttingPlaneFragement.glsl). The LocalZcomp represents the z-value of the given vector in the view frustum of the cubemap. The value is mapped to [0,1] through usage perspective division.

2.2.3 Shadows from cut surface

In order to keep the illusion of a solid object, the surface left by the cutting plane needs to cast shadows on the backside of the object. In order to do this, I calculate the intersection point in the light projection space for every back facing fragment to the light position for point light, and towards the light direction for directional light. For it to be pack properly, I convert the z-value of the intersection point to the range of [0,1], by subtracting the near-plane and dividing by the far-plane. This way, the back faces cast shadows as if they had the planes depth value in the light projection space.

3 External references

3.1 Plane equations

http://www.songho.ca/math/plane/plane.html

3.2 Omnidirectionl shadow maps

https://stackoverflow.com/questions/34639572/render-to-cubemap-tutorial-in-webgl
https://stackoverflow.com/questions/10786951/omnidirectional-shadow-mapping-with-depth-cubemap